Patrycja Kozak

200831

Tytuł pracy

Tytuł pracy w języku angielskim

Praca dyplomowa na kierunku:

Informatyka

Praca wykonana pod kierunkiem   
dr inż. Artur Krupa

Instytut Informatyki Technicznej

Katedra Sztucznej Inteligencji

Warszawa, rok 2024

**Oświadczenie Promotora pracy**

Oświadczam, że niniejsza praca została przygotowana pod moim kierunkiem   
i stwierdzam, że spełnia warunki do przedstawienia tej pracy w postępowaniu   
o nadanie tytułu zawodowego.

Data .................................... Podpis promotora ...................................................

**Oświadczenie autora pracy**

Świadom odpowiedzialności prawnej, w tym odpowiedzialności karnej za złożenie fałszywego oświadczenia, oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie i nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami prawa, w szczególności ustawą z dnia 4 lutego 1994 r. o prawie autorskim i prawach pokrewnych (Dz.U. 2019 poz. 1231 z późn. zm.).

Oświadczam, że przedstawiona praca nie była wcześniej podstawą żadnej procedury związanej z nadaniem dyplomu lub uzyskaniem tytułu zawodowego.

Oświadczam, że niniejsza wersja pracy jest identyczna z załączoną wersją elektroniczną. Przyjmuję do wiadomości, że praca dyplomowa poddana zostanie procedurze antyplagiatowej.

Data ..................................... Podpis autora pracy .............................................

**Streszczenie**

**Tytuł Pracy dyplomowej**

Streszczenie

Słowa kluczowe – monolit, mikroserwisy

**Summary**

**Tytuł Pracy w języku ang.**

Treść w języku angielskim

Keywords – …..

Spis treści

[**Wstęp** 9](#_Toc184768313)

[**1** **Przegląd literatury** 11](#_Toc184768314)

[**1.1** **Charakterystyka architektury monolitycznej** 11](#_Toc184768315)

[**1.2** **Charakterystyka architektury mikroserwis** 11](#_Toc184768316)

[**1.3** **Porównanie architektury monolitycznej oraz mikroserwisów** 12](#_Toc184768317)

[**2** **Wykorzystane rozwiązania** 15](#_Toc184768318)

[**2.1** **Języki programowania** 15](#_Toc184768319)

[**2.2** **Frameworki oraz biblioteki** 15](#_Toc184768320)

[**2.3** **Bazy danych i messaging** 16](#_Toc184768321)

[**2.4** **Konteneryzacja i orkiestracja** 16](#_Toc184768322)

[**2.5** **API Gateway** 17](#_Toc184768323)

[**3** **Implementacja** 19](#_Toc184768324)

[**3.1** **System w architekturze monolitycznej** 19](#_Toc184768325)

[3.1.1 Projekt systemu 19](#_Toc184768326)

[3.1.2 Szczegóły Implementacji 19](#_Toc184768327)

[3.1.3 Napotkanie wyzwania 19](#_Toc184768328)

[3.1.4 Metryki wydajnościowe 19](#_Toc184768329)

[**3.2** **System w architekturze mikroserwisów** 19](#_Toc184768330)

[3.2.1 Dekompozycja na mikroserwisy 19](#_Toc184768331)

[3.2.2 Komunikacja miedzy serwisami 19](#_Toc184768332)

[3.2.3 Zarzadzanie danymi 19](#_Toc184768333)

[3.2.4 Obsługa błędów i odporność na awarie 19](#_Toc184768334)

[3.2.5 Metryki wydajnościowe 19](#_Toc184768335)

[**4** **Podsumowanie i omówienie wyników** 20](#_Toc184768336)

[**5** **Bibliografia** 20](#_Toc184768337)

# **Wstęp**

W ciągu ostatnich lat na znaczeniu zyskały głównie dwie koncepcje architektoniczne: architektura monolityczna oraz mikroserwisowa. Wybór odpowiedniego podejścia może mieć kluczowe znaczenie dla sukcesu aplikacji, wpływając na elastyczność, skalowalność oraz łatwość utrzymania systemu.

Architektura monolityczna, będąca tradycyjnym podejściem do projektowania aplikacji, zakłada, że wszystkie komponenty systemu są zintegrowane w jedną całość. Architektura mikroserwisowa proponuje podział systemu na niezależne, współpracujące ze sobą serwisy. Każdy mikroserwis realizuje określoną funkcjonalność i może być rozwijany, wdrażany oraz skalowany niezależnie od innych

Wiele znanych firm rozpoczynało tworzenie swoich systemów od architektury monolitycznej, co w początkowych etapach rozwoju było rozwiązaniem prostym, logicznym oraz dającym zadowalający efekt. Przykładami takich firm są Netflix, Amazon, GitHub, Atlassian czy Basecamp [1]. Firma Netflix była jednym z pierwszych przedsiębiorstw, które zdało sobie sprawę, że architektura monolityczna może być niewystarczająca, i może nie być najlepszym rozwiązaniem w przypadku złożonych aplikacji. Architekci Netflixa, przez pojedynczy błąd z 2008 roku, który spowodował masową utratę danych i doprowadził do kilkudniowego przestoju, zdecydowali się przenieść swoją aplikację z architektury monolitycznej do mikroserwisów opartych na chmurze AWS [2]. Uznano, że migracja architektury, usprawni dostępność oraz skalowalność systemu. Podzielenie systemu na 700 mikroserwisów, z których każdy odpowiedzialny był za jedną funkcjonalność, umożliwiało dokonywanie zmian w dowolnej części systemu, mając pewność, że aplikacja nie zostanie wyłączona. Pomogło to w zminimalizowaniu braków dostępności do serwisu.

Wraz z dynamicznym rozwojem technologii oraz rosnącymi wymaganiami i potrzebami współczesnych aplikacji, pojawia się konieczność przeanalizowania, a także porównania architektury mikroserwisów oraz monolitycznej. Tradycyjne już podejście monolityczne, mimo swojej prostoty, często może nie sprostać wyzwaniom jakie może mieć aplikacja. Może ona okazać się niewystarczająca pod względem skalowalności, elastyczności czy szybkości wdrażania nowych, potrzebnych funkcji. Użycie architektury mikroserwisów, która pozwala na dekompozycję aplikacji na niezależnie rozwijalne komponenty, wymaga dokładnego zrozumienia, kiedy i dlaczego warto użyć właśnie jej, jako rozwiązania.

Analiza ta może być szczególnie istotna dla podmiotów stojących przed decyzją o modernizacji swojej architektury. Porównanie tych dwóch podejść pozwoli również zidentyfikować najlepsze praktyki oraz potencjalne pułapki, co jest kluczowe dla skutecznego wdrożenia nowoczesnych rozwiązań technologicznych.

Celem niniejszej pracy magisterskiej jest przeprowadzenie analizy porównawczej architektury monolitycznej oraz mikroserwisowej w kontekście ich odpowiedniości w różnych scenariuszach wdrożenia aplikacji. Proces badawczy będzie obejmował aspekty techniczne oraz funkcjonalne obu podejść, wraz z uwzględnieniem ich wpływu na skalowalność, wydajność, elastyczność, łatwość zarządzania oraz utrzymania systemu. Dodatkowymi aspektami porównawczymi będzie utylizacja zasobów oraz bezpieczeństwo. Praca ta będzie również skupiać się na identyfikacji sytuacji, w których każda z analizowanych architektur może być preferowana. Dodatkowo praca ta ma na celu dostarczenie praktycznych wskazówek dla specjalistów zajmującymi się architekturą oprogramowania w celu wyboru tej odpowiedniej w zależności od kontekstu projektu oraz jego wymagań.

**[TODO: OPISAC UKLAD JAK WSZYSTKO GOTOWE]**

# **Przegląd literatury**

## **Charakterystyka architektury monolitycznej**

Architektura monolityczna jest tradycyjnym podejściem do projektowania i budowania systemów informatycznych, w którym cała aplikacja funkcjonuje jako jednolity i nierozerwalny blok kodu. [3] W tego rodzaju architekturze wszystkie moduły systemu, takie jak interfejs użytkownika, logika biznesowa i warstwa danych, są ze sobą ściśle powiązane i uruchamiane wspólnie w ramach jednego procesu. Chris Richardson w swojej książce [4] wskazuje, że ta cecha wynika z fundamentalnego założenia monolitu - wszystkie funkcjonalności są kompilowane do jednego wykonywalnego pliku lub pakietu. Zintegrowana struktura monolitu sprawia, że aplikacja jest budowana, testowana i wdrażana jako jedna całość, co przekłada się na prostotę w początkowej fazie projektu.

Jedną z kluczowych zalet architektury monolitycznej jest jej prostota koncepcyjna i implementacyjna. Vaughn Vernon podkreśla, że cała aplikacja funkcjonująca w ramach jednego środowiska znacząco upraszcza procesy projektowania i implementacji, szczególnie dla małych zespołów programistycznych. Programiści mogą skupić się na rozwoju funkcjonalności bez konieczności zarządzania złożoną infrastrukturą czy komunikacją między komponentami. [5]

Architektura monolityczna ma jednak swoje ograniczenia, które stają się widoczne wraz ze wzrostem złożoności aplikacji. Jej skalowanie jest trudne i kosztowne, ponieważ wymaga skalowania całej aplikacji, nawet jeśli tylko jedna jej część wymaga większej wydajności. Utrzymanie staje się coraz bardziej problematyczne, ponieważ zależności między modułami sprawiają, że każda zmiana może wywołać nieprzewidziane problemy w innych częściach systemu. Dodatkowo, awaria jednego modułu może unieruchomić całą aplikację, co obniża niezawodność rozwiązania. Architektura monolityczna najlepiej sprawdza się w mniejszych projektach, które nie wymagają dużej skalowalności, takich jak wewnętrzne systemy ERP [6] czy proste aplikacje e-commerce.

## **Charakterystyka architektury mikroserwis**

Architektura mikroserwisowa to nowoczesne podejście do projektowania aplikacji, które zakłada podział systemu na wiele niezależnych usług, z których każda odpowiada za realizację jednej konkretnej funkcjonalności. Mikroserwisy działają jako odrębne jednostki, które mogą być rozwijane, wdrażane i skalowane niezależnie od siebie. Komunikacja między nimi odbywa się za pomocą lekkich protokołów, takich jak HTTP, gRPC czy komunikaty w kolejkach wiadomości. [7]

Główną zaletą architektury mikroserwisowej jest jej elastyczność i skalowalność. Dzięki niezależności poszczególnych usług można je skalować w zależności od obciążenia, co pozwala na efektywniejsze wykorzystanie zasobów. Dodatkowo, możliwość niezależnego wdrażania poszczególnych mikroserwisów przyspiesza wprowadzanie zmian i ułatwia dodawanie nowych funkcjonalności bez ryzyka destabilizacji całego systemu. Awaria jednego mikroserwisu zazwyczaj nie wpływa na działanie pozostałych, co zwiększa niezawodność aplikacji jako całości.

Architektura mikroserwisowa wymaga jednak większych nakładów na zarządzanie i utrzymanie. Rozproszona natura systemu sprawia, że niezbędne są zaawansowane narzędzia do monitorowania, orkiestracji (np. Kubernetes) i zarządzania komunikacją między usługami. Wdrożenie mikroserwisów wymaga także bardziej złożonej infrastruktury, co wiąże się z wyższymi kosztami początkowymi. Kolejnym wyzwaniem jest testowanie aplikacji, ponieważ integracja niezależnych usług może generować nieoczekiwane problemy. [8]

Architektura mikroserwisowa znajduje zastosowanie głównie w dużych, złożonych systemach, które wymagają skalowalności, niezawodności i elastyczności. Jest to popularne rozwiązanie w platformach streamingowych, takich jak Netflix, czy dużych serwisach e-commerce, takich jak Amazon, które muszą obsługiwać miliony użytkowników w sposób płynny i niezawodny. [9]

Wybór architektury mikroserwisowej pozwala na większą adaptacyjność do dynamicznych wymagań, jednak wiąże się z wyższymi kosztami i większą złożonością zarządzania.

## **Porównanie architektury monolitycznej oraz mikroserwisów**

W artykule [10] autor zgłębia się w długo toczącą się debatę pomiędzy architekturami monolitów oraz mikroserwisów. Omówione zostają powszechnie błędne przekonania na temat architektury mikroserwisów, twierdząc że nie są one cudownym środkiem ale raczej narzędziem do zarządzania złożonością nowoczesnych aplikacji. Autor podkreśla, że wszystkie aplikacje nieuchronnie stają się złożone, a mikroserwisy oferują sposób na złagodzenie zaistniałej złożoności poprzez możliwość podzielenia jej na mniejsze, oraz łatwiejsze do zarządzania fragmenty. Jednym z kluczowych argumentów autor uznał to, że rozmiar mikroserwisu ma mniejsze znaczenie niż jego skuteczność w zaspokajaniu konkretnych potrzeb aplikacji. Zamiast skupiać wyłącznie na tworzeniu jak najmniejszych usług, sugeruje on tworzenie serwisów które zapewniają równowagę pomiędzy złożonością a łatwością zarządzania. Ponadto w artykule podważono koncepcję dychotomii[[1]](#footnote-1) pomiędzy monolitami a mikroserwisami, przez co zaproponowany został model hybrydowy, łączący elementy obu architektur. Autor sugeruje, że taki model odzwierciedla rzeczywistość wielu aplikacji, w których współistnieje mieszanka komponentów opartych na podejściu monolitycznym oraz mikroserwisowym. Omówiona została także ewolucja mikroserwisów na przestrzeni czasu, gdzie wywnioskowane zostało, że postęp w narzędziach i platformach sprawił, że ten rodzaj architektury stał się dużo bardziej dostępny i przyjazny.

W ramach pracy [11] autorzy porównali wydajność samodzielnie opracowanego systemu opartego na mikroserwisach z jego monolitycznym odpowiednikiem, zbadali także wpływ konteneryzacji na wydajność systemu. Korzystając z JMeter [12] do symulacji użytkowników, stwierdzono, że system monolityczny miał krótszy czas reakcji oraz większą przepustowość, zużywając mniej pamięci RAM oraz procesora w porównaniu do systemu mikroserwisów. Konteneryzacja systemu, przy użyciu Dockera [13] nie zmniejszyła wydajności, wręcz wykazała poprawę zużycia pamięci RAM, procesora, przepustowości oraz zauważono mniejsze opóźnienia. Wykorzystanie Dockera na systemie opartym o mikroserwisy skutkowało podobnymi opóźnieniami, ale zwiększyło przepustowość nawet o 15% przy dużym obciążeniu systemu, jednakże kosztem użycia większej ilości pamięci RAM i procesora.

Kolejny artykuł [14] ocenia takie wskaźniki jak czas reakcji oraz przepustowość zarówno przy wysokim jak i niskim obciążeniu. Dodatkowo autorzy porównali skuteczność różnych technologii wykrywania usług (Service discovery technologies) [15] Eureka oraz Consul, dla architektur mikroserwisowych. Ich ustalenia wskazują, że przy typowych obciążeniach obie te architektury wykazują podobny poziom wydajności. Jednakże różnice stają się widoczne wraz ze wzrostem obciążenia, gdzie aplikacje monolityczne wykazują tendencję do wykazywania większej wydajności. W artykule tym podkreślono, że użycie architektury monolitycznej jest zalecane w przypadku, gdy celem programisty jest to by aplikacja obsługiwała żądania szybciej. Warto zauważyć, że pod względem przepustowości oraz liczby obsługiwanych żądań na sekundę, mikroserwisy wykorzystujące Consul do wykrywania usług, były o 4% lepsze niż te używające Eureki.

Autorzy [16] porównując architketury monolitów oraz mikroserwisów w swojej pracy zauważają, że architektura monolityczna zazwyczaj oferuje lepszą wydajność na pojedynczej maszynie niż mikroserwisy, ze względu na dodatkowe obciążenie związane z przekazywaniem żądań między komponentami.

W pracy [17] wykonano porównanie obu architektur przy użyciu narzędzia do testowania obciążenia Gatling [18]. Badania te zostały wykonane na maszynie z systemem operacyjnym Ubuntu 18.04.2 LTS, a aplikacje wdrożone zostały przy pomocy Dockera. Praca ta porównuje te architektury pod względem takim parametrów jak: wydajność i czas odpowiedzi. Testy obciążeniowe wykazały ze mikroserwisy są bardziej wydajne, gdy aplikacja musi przetworzyć więcej requestów. Natomiast monolity mogą być dobrym rozwiązaniem, w przypadku aplikacji o mniejszym obciążeniu, przy czym dodatkowo są prostsze do stworzenia. Autorzy wspominają, że wybór architektury powinien być zdefiniowany przez wymagania biznesowe, tak aby spełniało ono wymagania inwestorów.

## 

# **Wykorzystane rozwiązania**

Wybór opisanych poniżej technologii został podyktowany zarówno zainteresowaniami autora, jak i jego doświadczeniem w pracy z nimi. W celu zapewnienia wysokiej jakości kodu, opierano się na wskazówkach i rekomendacjach zawartych w książkach [19] [20]

## **Języki programowania**

**.Net oraz C#**

Platforma .NET to darmowe, otwarte środowisko programistyczne, które umożliwia tworzenie i uruchamianie aplikacji na różnych, popularnych platformach. W momencie rozpoczęcia pracy nad aplikacją najnowszą dostępną wersją .NET była wersja 8. [21] Język C# jest nowoczesnym, obiektowym językiem programowania, w którym kod jest kompilowany do CIL (Common Intermediate Language).

**Typescript**

Typescript to silnie typowany język programowania, który rozszerza możliwości JavaScript poprzez wprowadzenie statycznych typów. Kompiluje się do standardowego JavaScript, dzięki czemu może być używany we wszystkich nowoczesnych przeglądarkach oraz środowiskach uruchomieniowych. [22]

## **Frameworki oraz biblioteki**

**MassTransit**

MassTransit jest oprogramowaniem typu open-source dla platformy .NET, służącym do implementacji wzorców komunikacji między usługami. Framework ten umożliwia asynchroniczną komunikację oraz integrację między systemami przy użyciu wiadomości. Rozwiązanie to jest stosowanie w tworzeniu rozwiązań opartych na mikrousługach. [23]

**Vue.js 3**

Vue to progresywny framework JavaScript do budowania interfejsów użytkownika oraz aplikacji jednostronicowych (SPA – Single Page Applications). Cechuje się wydajnością, łatwością użycia oraz elastycznością. [24]

## **Bazy danych i messaging**

**Microsoft SQL Server**

Microsoft SQL Server relacyjny system zarządzania bazą danych (RDBMS) stworzony przez Microsoft. Charakteryzuje się wysoką wydajnością, niezawodnością i skalowalnością. Wykorzystuje język T-SQL do zarządzania danymi i zapytań. SQL Server doskonale integruje się z technologiami Microsoft, co czyni go popularnym rozwiązaniem w aplikacjach biznesowych i korporacyjnych [25]

**RabbitMQ**

RabbitMQ jest systemem kolejkowania wiadomości (message broker) opartym na protokole AMQP, czyli standardowym protokole komunikacyjnym, służącym do niezawodnego przesyłania wiadomości między systemami. [26] Umożliwia asynchroniczną komunikację między usługami, zapewniając niezawodne przesyłanie, kolejkowanie i odbieranie wiadomości. Charakteryzuje się wysoką wydajnością, skalowalnością i wsparciem dla wielu języków programowania, dzięki czemu jest często stosowany w systemach rozproszonych i mikroserwisach. [27]

## **Konteneryzacja i orkiestracja**

**Docker**

Docker jest platformą do tworzenia, uruchamiania i zarządzania kontenerami, które izolują aplikacje wraz ze wszystkimi ich zależnościami. Dzięki temu zapewnia spójność działania aplikacji w różnych środowiskach, takich jak lokalne maszyny, serwery czy chmura. Docker jest lekki, szybki i umożliwia efektywne zarządzanie zasobami. [28]

**Docker Compose**

Docker Compose jest narzędziem umożliwiającym definiowanie i uruchamianie aplikacji składających się z wielu kontenerów za pomocą jednego pliku konfiguracyjnego YAML. Ułatwia to zarządzanie skomplikowanymi środowiskami, w których poszczególne usługi działają w oddzielnych kontenerach. Narzędzie to pozwala na szybkie wdrażanie, skalowanie i testowanie aplikacji wielokontenerowych. [29]

## **API Gateway**

API Gateway jest pośrednikiem zarządzającym ruchem między klientami, a usługami w systemach opartych na mikroserwisach. Służy jako pojedynczy punkt wejścia dla żądań API, umożliwiając tym przekierowanie do odpowiednich usług. [30]

**Ocelot**

Ocelot jest to API Gateway dla platformy .NET, który służy jako brama wejściowa do mikroserwisów, zarządzając routingiem żądań, agregacją, loadbalancingiem oraz dostarczając funkcje takie jak cache'owanie czy rate limiting. Jest to narzędzie open-source, które pozwala na łatwe konfigurowanie tras za pomocą pliku JSON, dzięki czemu można w prosty sposób przekierowywać ruch z jednego punktu wejścia do różnych mikroserwisów. [31]

# **Implementacja**

Implementacja została zrealizowana etapowo. Pierwszym krokiem została stworzona wersja monolityczna, która pozwoliła na szybkie opracowanie podstawowej funkcjonalności oraz przetestowanie głównych założeń systemu. Model ten umożliwił na łatwiejsze zrozumienie logiki biznesowej oraz eliminację potencjalnych błędów na wczesnym etapie projektu. Następnie, zgodnie z podejściem rekomendowanym w literaturze [32], przeprowadzona została dekompozycja monolitu na niezależne mikroserwisy. Proces dekompozycji polegał na identyfikacji kluczowych komponentów systemu, które następnie zostały wydzielone jako odrębne serwisy.

## **System w architekturze monolitycznej**

### Projekt systemu

### Szczegóły Implementacji

### Napotkanie wyzwania

### Metryki wydajnościowe

## **System w architekturze mikroserwisów**

### Dekompozycja na mikroserwisy

### Komunikacja miedzy serwisami

### Zarzadzanie danymi

### Obsługa błędów i odporność na awarie

### Metryki wydajnościowe

# **Podsumowanie i omówienie wyników**

# **Bibliografia**

|  |  |
| --- | --- |
| [1] | „https://medium.com/@sadhnajuhirdeen/what-are-the-big-companies-still-uses-monolithic-architecture-2555a93345f1,” [Online]. [Data uzyskania dostępu: 1 05 2024]. |
| [2] | „https://www.techtarget.com/searchaws/definition/Amazon-Web-Services,” [Online]. [Data uzyskania dostępu: 05 2024]. |
| [3] | „https://beecommerce.pl/blog/co-to-jest-aplikacja-monolityczna,” 05 12 2024. [Online]. |
| [4] | C. Richardson, Microservices Pattern ith examples in Java. |
| [5] | V. Vernon, Implementing Domain-Driven Design. |
| [6] | „https://www.sap.com/poland/products/erp/what-is-erp.html,” [Online]. [Data uzyskania dostępu: 05 12 2024]. |
| [7] | „https://www.atlassian.com/pl/microservices/microservices-architecture,” [Online]. |
| [8] | B. Brendan, Designing Distributed Systems: Patterns and Paradigms for Scalable, Reliable Services. |
| [9] | „https://keen.io/blog/architecture-of-giants-data-stacks-at-facebook-netflix-airbnb-and-pinterest/,” [Online]. [Data uzyskania dostępu: 05 12 2024]. |
| [10] | A. Davis, „https://www.infoq.com/articles/monolith-versus-microservices/,” 05 2024. [Online]. |
| [11] | R. Flygare i A. Holmqvist, „Performance characteristics between monolithic and microservice-based systems,” [Online]. Available: https://www.diva-portal.org/smash/record.jsf?pid=diva2%3A1119785&dswid=-457. |
| [12] | „https://jmeter.apache.org/,” [Online]. [Data uzyskania dostępu: 05 2024]. |
| [13] | „https://docs.docker.com/get-started/overview/,” [Online]. [Data uzyskania dostępu: 05 2024]. |
| [14] | O. Al-Debagy i P. Martinek, „A Comparative Review of Microservices and Monolithic Architectures”. |
| [15] | „https://www.nginx.com/blog/service-discovery-in-a-microservices-architecture/,” [Online]. [Data uzyskania dostępu: 05 2024]. |
| [16] | A. O. A. P. G. Blinowski, „https://www.semanticscholar.org/paper/Monolithic-vs.-Microservice-Architecture%3A-A-and-Blinowski-Ojdowska/31a9d5d7286b24d5d2a99af005dca7a814640aec,” [Online]. [Data uzyskania dostępu: 05 2024]. |
| [17] | W. Z. Konrad Gos, „https://scholar.google.pl/citations?view\_op=view\_citation&hl=pl&user=h0ikHxkAAAAJ&citation\_for\_view=h0ikHxkAAAAJ:bnK-pcrLprsC,” [Online]. [Data uzyskania dostępu: 05 2024]. |
| [18] | „https://docs.gatling.io/,” [Online]. [Data uzyskania dostępu: 05 2024]. |
| [19] | R. C. Martin, „Czysty kod. Podręcznik dobrego programisty”. |
| [20] | R. C. Martin, „Czysta architektura”. |
| [21] | „https://learn.microsoft.com/pl-pl/dotnet/core/whats-new/dotnet-8/overview,” [Online]. [Data uzyskania dostępu: 05 12 2024]. |
| [22] | „https://www.typescriptlang.org/,” [Online]. [Data uzyskania dostępu: 05 12 2024]. |
| [23] | „https://masstransit.io/introduction,” [Online]. [Data uzyskania dostępu: 05 12 2024]. |
| [24] | „https://vuejs.org/,” [Online]. [Data uzyskania dostępu: 05 12 2024]. |
| [25] | „https://learn.microsoft.com/en-us/sql/sql-server/what-is-sql-server?view=sql-server-ver16,” [Online]. [Data uzyskania dostępu: 05 12 2024]. |
| [26] | „https://boringowl.io/blog/wprowadzenie-do-amqp-advanced-message-queuing-protocol,” [Online]. Available: 05. |
| [27] | „https://www.rabbitmq.com/,” [Online]. [Data uzyskania dostępu: 05 12 2024]. |
| [28] | „https://britenet.eu/pl/blog-post/czym-jest-docker-i-jakie-sa-korzysci-z-jego-uzywania/,” [Online]. [Data uzyskania dostępu: 05 12 2024]. |
| [29] | „https://docs.docker.com/compose/,” [Online]. [Data uzyskania dostępu: 05 12 2024]. |
| [30] | „https://learn.microsoft.com/pl-pl/dotnet/architecture/microservices/architect-microservice-container-applications/direct-client-to-microservice-communication-versus-the-api-gateway-pattern,” [Online]. [Data uzyskania dostępu: 05 12 2024]. |
| [31] | „https://ocelot.readthedocs.io/en/latest/introduction/bigpicture.html,” [Online]. [Data uzyskania dostępu: 05 12 2024]. |
| [32] | „Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith,” [Online]. [Data uzyskania dostępu: 05 12 2024]. |
| [33] | „https://microservices.io/patterns/monolithic.html,” [Online]. Available: https://microservices.io/patterns/monolithic.html. [Data uzyskania dostępu: 10 12 2024]. |
| [34] | „https://dzone.com/articles/the-evolution-of-distributed-systems,” [Online]. [Data uzyskania dostępu: 05 12 2024]. |

Wyrażam zgodę na udostępnienie mojej pracy w czytelniach Biblioteki SGGW   
w tym w Archiwum Prac Dyplomowych SGGW

.................................................................

*(czytelny podpis autora pracy)*

1. Dychotomia - podział jakiejś całości na dwie różniące się zasadniczo części; też: ta różnica [↑](#footnote-ref-1)